

## 10. Standard, linguaggi e progettazione di database a oggetti

Definire uno standard per un particolare tipo di database system è importante, poiché offre tre vantaggi:

- Portabilità - È la capacità di eseguire un'applicazione su sistemi differenti con modifiche minime.
- Interoperabilità - È la capacità di un'applicazione di accedere a più sistemi distinti.
- Confrontabilità di prodotti commerciali - È la capacità di confrontare prodotti commerciali determinando quali parti dello standard sono supportate da ogni prodotto.

Un consorzio di produttori di DBMS O-O, chiamato ODMG (Object Database Management Group) ha proposto un standard, detto ODMG 1.0 poi rivisto nell'ODMG 2.0. Lo standard è composto da quattro parti:

1. Il modello ad oggetti
2. Il linguaggio di definizione degli oggetti (ODL)
3. Il linguaggio di interrogazione degli oggetti (OQL)
4. I binding con i linguaggi di programmazione: sono stati definiti binding con Java, C++ e SMALLTALK.

### ***Panoramica sul modello a oggetti di ODMG***

L'Object Model di ODMG è il paradigma su cui sono basati i linguaggi di definizione (ODL) ed interrogazione (OQL). Il modello comprende tipi di dati, costruttori di tipi ed altri concetti necessari per specificare lo schema del db. Il modello dunque riveste per i database ad oggetti lo stesso ruolo svolto dal SQL nei db relazionali.

Oggetti e letterali sono i costrutti di base del modello ad oggetti. Entrambi possono avere delle strutture complesse. Le differenze sono che un oggetto ha sia un identificatore che uno stato (valore corrente), mentre un letterale ha solo un valore. Lo stato di un oggetto può cambiare nel tempo, mentre un letterale è sostanzialmente un valore costante.

Un oggetto è descritto da quattro caratteristiche:

1. **Identificatore** - ogni oggetto deve avere un identificatore, univoco in tutto il sistema.
2. **Nome** - opzionalmente, un oggetto può avere un nome (ed essere usato come entry point: ritrovando tali oggetti attraverso il nome unico, si possono reperire altri oggetti ad essi collegati).
3. **LifeTime** - specifica se l'oggetto è persistente o transiente.
4. **Struttura** - specifica se l'oggetto è atomico o è una collezione.

Nel modello ad oggetti, un letterale è un valore senza OID con una struttura semplice o complessa. Sono ammessi tre tipi di letterali:

- **Atomici** - Sono i tipi di dati di base, quali Char, Long, Boolean, ecc.
- **Strutturati** - Sono delle strutture complesse, predefinite (es. Date, Time, Intervalecc.) o definite dall'utente.

- **Collezioni** - Sono una collezione (senza ID) di oggetti o di valori. Possono essere Set, Bag, List ed Array.

Un'interfaccia è una definizione di metodi astratti (cioè senza implementazione), di cui sono specificati gli attributi visibili, le relazioni, e vi sono solo la signature dei metodi. Una interfaccia non è istanziabile (poiché manca l'implementazione delle operazioni). Le interfacce sono usate per definire operazioni comuni a tutte le classi che ne derivano (polimorfismo).

Una classe è una specifica di metodi e stati: È istanziabile (è possibile creare istanze di oggetti corrispondenti alla definizione della classe).

Esistono due tipi di ereditarietà: ereditarietà di Interfaccia, specificata col simbolo “:”, il supertipo deve essere un'interfaccia ed il sottotipo una classe o un'interfaccia; e ereditarietà di classe, specificata con la keyword **extends**; è utilizzata esclusivamente tra classi. Non è permessa ereditarietà di classe multipla, mentre è permessa ereditarietà di interfaccia multipla.

In ODMG tutti gli oggetti ereditano l'interfaccia di base Object:

```
interface Object {
    ...
    boolean    same_as(in Object other_object);
    Object     copy();
    void       delete();
};
```

Vediamo ora alcuni esempi di oggetti definiti via ODMG.

### ***Esempio.***

```
interface Time : Object {
    ...
    unsigned short  hour();
    unsigned short  minute();
    unsigned short  second();
    unsigned short  millisecond();
    ...
    boolean         is_equal(in Time other_Time);
    boolean         is_greater(in Time other_Time);
    ...
    Time            add_interval(in Interval some_Interval);
    Time            subtract_interval(in Interval some_Interval);
    Interval        subtract_time(in Time other_Time);
};
```

*Illustrazione 1: Un esempio di definizione di interfaccia in ODMG.*

Le collezioni nel modello ad oggetti sono SET<T>, BAG<T>, LIST<T> e ARRAY<T>, dove **t** è il tipo di oggetti o valori della collezione. Un altro tipo di collezione è DICTIONARY<K,V>, che è un insieme di associazioni <k,v>, dove **k** è una chiave (un valore unico di ricerca) associata al valore **v**. Può essere usato per creare un indice su una collezione di valori. I valori letterali di tipo collezione specificano un valore che è una collezione di oggetti o valori ma che non possiede un OID.

**Esempio.**

```

interface Collection : Object {
  ...
  exception      ElementNotFound{any element; };
  unsigned long  cardinality();
  boolean        is_empty();
  ...
  boolean        contains_element(in any element);
  void           insert_element(in any element);
  void           remove_element(in any element)
                raises(ElementNotFound);
  Iterator       create_iterator(in boolean stable);
  ...
};

```

*Illustrazione 2: Definizione dell'interfaccia dell'oggetto Collection.*

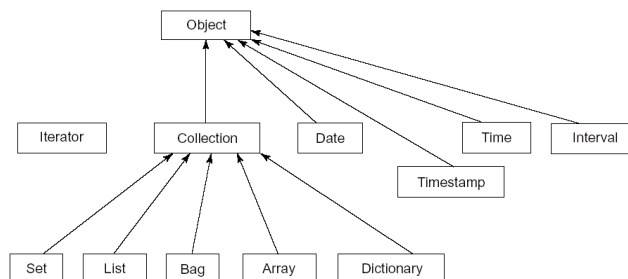
Dato un oggetto collection O, sono disponibili le seguenti operazioni predefinite (**dot notation**):

- O.cardinality()
- O.is\_empty()
- O.insert\_element(e)
- O.remove\_element(e)
- O.contains\_element(e)

(**arrow notation**):

- O → copy()
- O → same\_as(e)

Per gli altri tipi di oggetti collection (Set, Bag, List, Array e Dictionary), le rispettive interfacce predefinite in ODMG mettono a disposizione tutti i classici metodi propri di queste strutture dati. Gli attributi che hanno una struttura complessa sono definiti attraverso **struct**: corrisponde al costruttore di tipo *tupla*.



*Illustrazione 3: Gerarchia di ereditarietà per le interfacce predefinite.*

Nel modello ad oggetti, qualsiasi oggetto definito dall'utente non di tipo collection, è di tipo atomico. Gli oggetti atomici sono specificati per mezzo della keyword **class**. Una definizione di classe comprende la specifica di proprietà ed operazioni: le proprietà si dividono in attributi e

relazioni.

Un attributo è una proprietà che descrive qualche aspetto dell'oggetto. Tipicamente sono dei letterali contenuti nell'oggetto. Una relazione è una proprietà che specifica i legami tra due oggetti del db. In ODMG sono esplicitate solo le relazioni binarie, rappresentate con riferimenti inversi.

Ogni tipo oggetto può avere un insieme di signature di operazioni, che specificano il nome dell'operazione, il tipo degli argomenti, il tipo restituito, eventuali eccezioni che possono occorrere durante l'esecuzione dell'operazione. I nomi delle operazioni devono essere univoci all'interno di una classe.

### Esempio 1.

```
class Employee
( extent all_employees
  key   ssn )
{
  attribute string          name;
  attribute string         ssn;
  attribute date           birthdate;
  attribute enum Gender{M, F} sex;
  attribute short         age;
  relationship Department works_for
                        inverse Department::has_emps;
  void      reassign_emp(in string new_dname)
            raises(dname_not_valid);
};
```

*Illustrazione 4: Definizione della classe Employee.*

La classe Employee contiene attributi, relazioni e operazioni.

### Esempio 2.

```
class Department
( extent all_departments
  key   dname, dnumber )
{
  attribute string          dname;
  attribute short          dnumber;
  attribute struct Dept_Mgr {Employee manager, date startdate} mgr;
  attribute set<string>    locations;
  attribute struct Projs {string projname, time weekly_hours} projs;
  relationship set<Employee> has_emps inverse Employee::works_for;
  void      add_emp(in string new_ename) raises(ename_not_valid);
  void      change_manager(in string new_mgr_name; in date startdate);
};
```

*Illustrazione 5: Definizione della classe Department.*

Da notare che la definizione di classe contiene attributi, attributi complessi (Dept\_Mgr), relazioni e operazioni.

Nel modello ad oggetti di ODMG 2.0 è possibile definire un **extent** (un contenitore) per ogni tipo di oggetto definito attraverso una dichiarazione di classe. L'extent ha un nome e si comporta come un oggetto set, contenente tutti gli oggetti *persistenti* di una determinata classe. Ad esempio, la classe Employee contiene un extent all\_employees, che è equivalente alla creazione di un tipo

Set<Employee>.

Gli extents sono anche usati per far valere relazioni di **set/subset**. Se due classi A e B hanno extents all\_A e all\_B, e la classe B è sottotipo di A, allora la collezione di oggetti in all\_B deve essere un sottoinsieme di quelli di all\_A. Questo vincolo è fatto valere dal database system.

Una classe con un extent può avere una o più chiavi. Una chiave è composta da una o più proprietà (attributi o relazioni), con valori univoci per ogni oggetto nell'extent.

Un factory object è un oggetto usato per creare altri oggetti, attraverso le operazioni che mette a disposizione.

```
interface ObjectFactory {
    Object      new();
};
```

L'interfaccia ObjectFactory ha una singola operazione, la **new()**, che restituisce un nuovo oggetto con un OID. ODMG 2.0 mette a disposizione una serie di factory predefinite:

```
interface DateFactory : ObjectFactory {
    exception      InvalidDate{};
    ...
    Date           calendar_date(   in unsigned short year,
                                   in unsigned short month,
                                   in unsigned short day)
                                   raises(InvalidDate);
    ...
    Date           current();
};
```

*Illustrazione 6: La factory predefinita per le date.*

Poiché un ODBMS può creare più db, il modello ad oggetti di ODMG mette a disposizione gli oggetti Database e DatabaseFactory:

```
interface DatabaseFactory {
    Database      new();
};

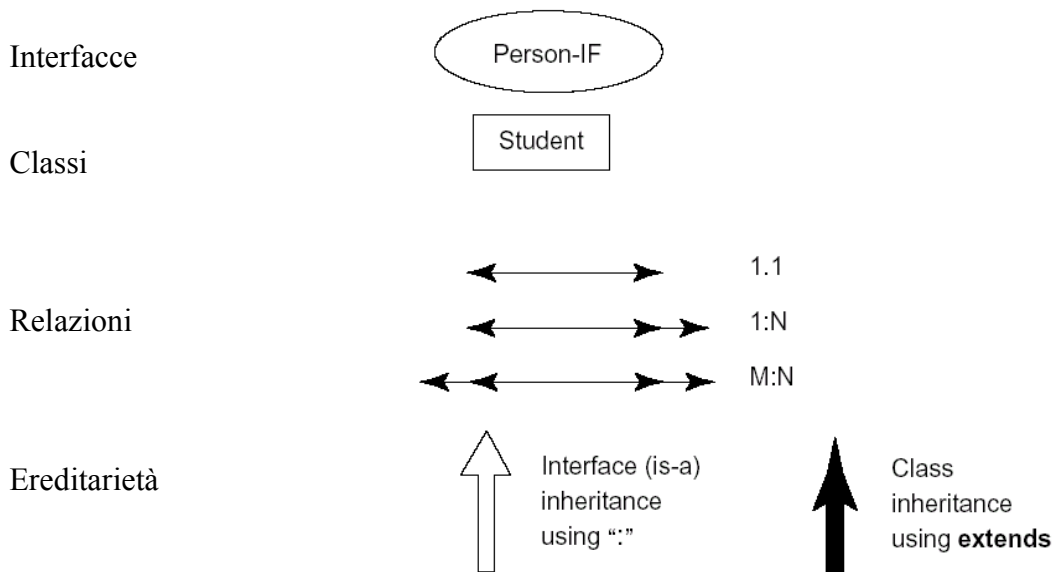
interface Database {
    void          open(in string database_name);
    void          close();
    void          bind(in any some_object, in string object_name);
    Object        unbind(in string name);
    Object        lookup(in string object_name)
                 raises(ElementNotFound);
    ...
};
```

## ***L'Object Definition Language***

L'ODL è progettato per supportare i costrutti semantici del modello ad oggetti di ODMG 2.0. È usato principalmente per definire classi ed interfacce. È indipendente dal linguaggio di programmazione utilizzato .

Dopo aver specificato lo schema del db in ODL, utilizzando specifici binding si determinano i mapping dei costrutti ODL sui costrutti del linguaggio di programmazione utilizzato.

La notazione grafica dell'ODL è così espressa:



**Esempio.**

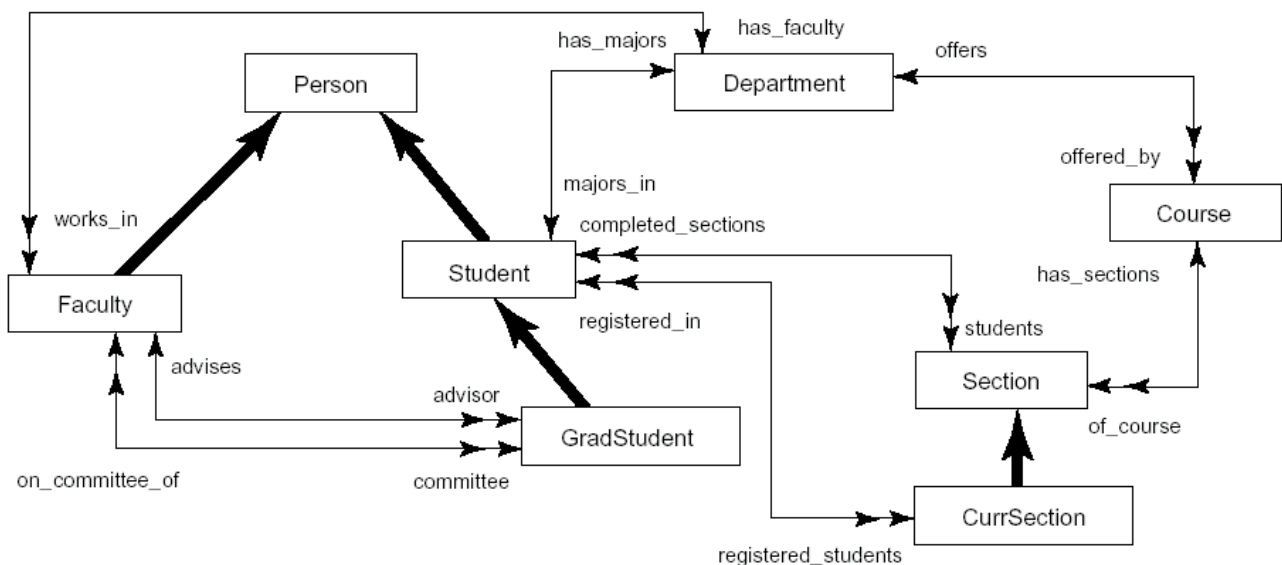


Illustrazione 7: Notazione grafica di parte dello schema di un database "University".

Vediamo adesso la traduzione in ODL di una parte dello schema University:

```

class Person
( extent persons
  key ssn )
{
  attribute struct Pname {string fname, string mname, string mname }
                                name;
  attribute string ssn;
  attribute date birthdate;
  attribute enum Gender{M, F} sex;
  attribute struct Address {short no, string street, short aptno,
string city, string state, short zip }
                                address;
  short age();
};

class Faculty extends Person
( extent faculty )
{
  attribute string rank;
  attribute float salary;
  attribute string office;
  attribute string phone;
  relationship Department works_in inverse Department::has_faculty;
  relationship set<GradStudent> advises inverse GradStudent::advisor;
  relationship set<GradStudent> on_committee_of
                                inverse GradStudent::committee;
  void give_raise(in float raise);
  void promote(in string new_rank);
};

```

vedi altri esempi di ereditarietà di interfacce alla slide 34-35 della lez 10.

## ***Object Query Language***

L'Object Query Language è il linguaggio di interrogazione proposto dal modello ad oggetti di ODMG. Poiché l'OQL è progettato per interagire fortemente con linguaggi di programmazione quali C++, SMALLTALK e Java, una query in OQL restituisce oggetti che possono essere facilmente tradotti nei tipi di tali linguaggi.

La sintassi di OQL è molto simile a quella di SQL, integrata di concetti propri di ODMG, quali OID, oggetti complessi, ereditarietà, polimorfismo, ecc.

Il costrutto di base di OQL per l'interrogazione dati è, come in SQL, **Select...From...Where**

### ***Esempio (Q0):***

```

Select d.dname
From d in departments
Where d.college = "Engeneering";

```

Per ogni query è richiesto un entry point. Questo può essere qualunque oggetto persistente con un nome: spesso l'entry point è l'extent della classe. In tal caso, l'entry point si riferisce ad una collezione di oggetti.

In presenza di collezioni, si definisce una variabile iteratore (d nell'esempio precedente Q0), che spazia su ogni oggetto della collezione.

La query, in genere, seleziona degli oggetti dalla collezione, in base alla clausola *where*. Il risultato di una query è:

- Un Bag, per gli statement **select... from**
- Un Set, per gli statement **select distinct... from**

La query Q0 restituisce un Bag<String>.

Più in generale, il risultato di una query può essere qualsiasi tipo ammesso nel modello ad oggetti di ODMG.

Una query non ha necessariamente la struttura Select... From... Where; qualsiasi nome persistente è una query, che restituisce il corrispondente oggetto persistente .

**Esempio (Q1):**

```
Departments
```

restituisce la collezione di tutti gli oggetti Department persistenti.

Determinato un entry point, si può usare il concetto di espressione di percorso per specificare un percorso fino agli attributi ed agli oggetti relati. Un'espressione di percorso parte da un oggetto persistente con nome (o un iteratore), ed è seguito da zero o più nomi di attributi/relazioni, usando la notazione punto.

**Esempio (Q2):**

```
csdepartment.chair.rank;
```

Una query OQL può restituire una struttura complessa, specificata nella query utilizzando la keyword **struct**.

**Esempio (Q4a):**

```
select struct
  (name:struct(last_name:s.name.lname,
              first_name:s.name.fname),
  degrees:(select struct (deg:d.degree,
                        yr: d.year,
                        college: d.college)
           from d in s.degrees
  )
  from s in csdepartment.chair.advises;
```

Il risultato della query Q4a è una collezione di struct (di primo livello), dove ogni struct ha due componenti: *name* e *degrees*. Il componente *name* è un'ulteriore struttura, composta da *last\_name* e *first\_name* (due stringhe). Il componente *degrees* è definita da una query incorporata, ed è esso stesso una collezione di struct (di secondo livello), contenente tre stringhe.

La specifica di viste in OQL usa il concetto di query con nome (named query). Si utilizza la keyword **define** per specificare un identificatore della query, che deve essere un nome univoco in tutto il db. Una vista può anche avere una serie di argomenti.

**Esempio (V1):**

```

define has_minors(deptname) as
select s
from s in students
where s.minors_in.dname = deptname;

```

Sappiamo che una query OQL restituisce una collezione di oggetti (bag, set o list): se la collezione contiene un solo oggetto, può essere utile ottenere direttamente il riferimento a tale oggetto, piuttosto che alla collezione. Ciò può essere effettuato per mezzo dell'operatore **element**.

**Esempio (Q6):**

```

element (select d
         from d in departments
         where d.dname= "Computer Science");

```

Dovendo quasi sempre gestire collezioni di oggetti, OQL mette a disposizione una serie di operazioni da applicare a tali strutture. Tra queste operazioni troviamo: operatori di aggregazione (min, max, count, sum e avg), quantificatori (esistenziali ed universali), operatori di appartenenza.

**Esempio (Q8):**

```

avg (select s.gpa
     from s in students
     where s.majors_in.dname = "Computer Science" and s.class = "senior");

```

Le funzioni di aggregazione possono essere usate su qualsiasi tipo, anche all'interno di una query.

**Esempio (Q9):**

```

select d.dname
from d in departments
where count (d.has_majors) > 100;

```

Le espressioni di appartenenza e quantificazione restituiscono un booleano. Sia **v** una variabile, **c** una collezione, **b** un'espressione booleana ed **e** un elemento di **c**:

- (**e in c**) è vera se *e* appartiene alla collezione *c*.
- (**for all v in c:b**) è vera se tutti gli elementi in *c* soddisfano *b*.
- (**exists v in c:b**) è vera se esiste almeno un elemento in *c* che soddisfa *b*.

**Esempio (Q10):**

```

select s.name.lname, s.name.fname
from s in students
where "Database System I" in
      (select c.cname
       from c in s.completed_sections.section.of_course)

```

Oltre all'uso dell'operatore in, Q10 mostra un modo semplice per specificare la clausola select di query che restituiscono una collezione di struct: Q10 restituisce un bag<struct<string,string>> .

Collezioni rappresentate da liste ed array hanno operazioni aggiuntive, quali il recupero dell'esimo, del primo o dell'ultimo elemento, estrazione di una sottocollezione, e concatenazione di liste.

**Esempio (Q14):**

```

first ( select struct (faculty:f.name.lname, salary:f.salary)
       from f in faculty
       order by f.salary desc);

```

La clausola group by di OQL, sebbene simile a quella di SQL, fornisce riferimenti espliciti alla collezione di oggetti in ogni gruppo o partizione.

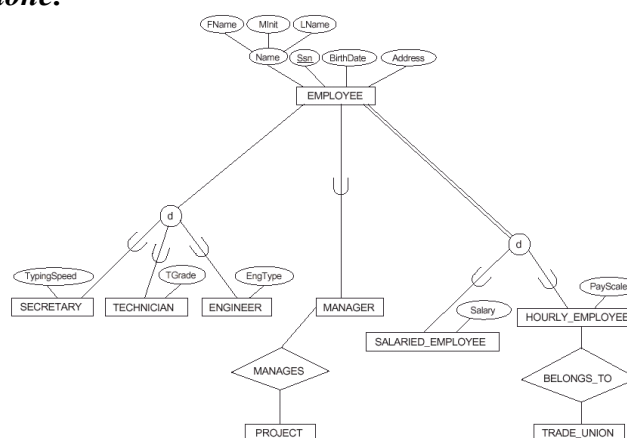
**Esempio (Q16):**

```

select struct(deptname, num_of_majors: count(partition) )
from s in students
group by deptname: s.majors_in.dname;

```

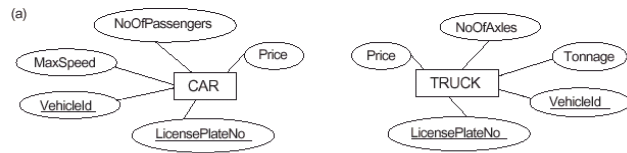
Il tipo restituito è un set<struct(dept\_name:string, num\_of\_majors:integer)> .

**Progettazione concettuale di un database a oggetti****Esempio di specializzazione:**

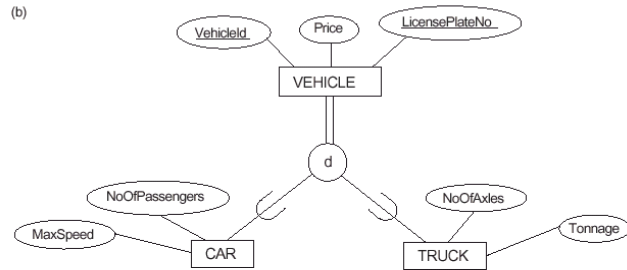
Tre specializzazioni di “Employee”: {Segretarie, Ingegneri, Tecnici} {Manager} {Impiegati fissi, Impiegati Part-Time}

**Esempio di generalizzazione.**

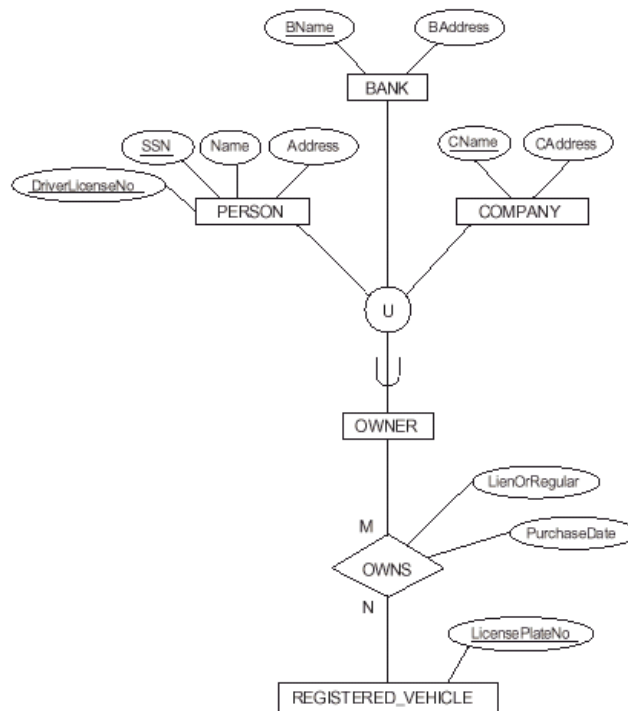
a) Due tipi di entità, CAR e TRUCK



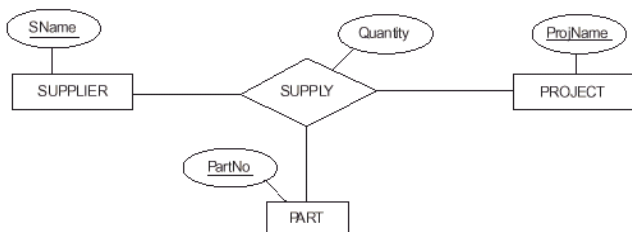
b) La loro generalizzazione nel tipo di entità VEHICLE



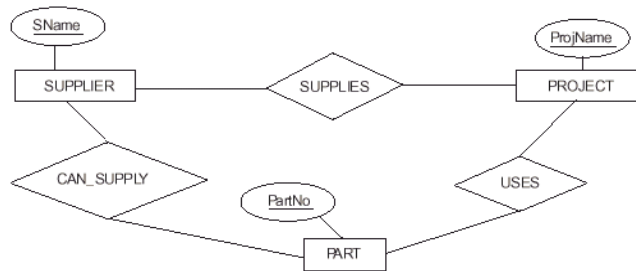
**Esempio: Tipi Unione e Categorie**



**Esempio: relazione con grado maggiore di 2**



tipo di relazione ternario



tre tipi di relazione binaria

Una delle principali differenze di progettazione tra database ad oggetti e database relazionali è nella gestione delle relazioni: nei db O-O le relazioni sono gestite con proprietà di relazione o attributi riferimento che includono gli OID degli oggetti relati. Questo porta a problemi con le relazioni n:m.

È possibile mappare in schemi O-O degli schemi EER non contenenti né categorie né relazioni con grado maggiore di 2. La procedura è composta da sette passi:

**Passo 1:** Creare una classe ODL per ogni tipo entità o sottoclasse EER. Il tipo della classe ODL dovrebbe includere tutti gli attributi della classe EER usando un costruttore di tupla al top level del tipo. Gli attributi multivalued sono dichiarati usando costruttori di set, bag o liste. Attributi composti sono mappati in un costruttore di tupla(struct). Dichiarare un extent per ogni classe, specificandone la chiave.

**Passo 2:** aggiungere proprietà di relazione o attributi di referenza per ogni relazione binaria nelle classi ODL che partecipano alla relazione. Gli attributi possono essere creati in una sola o in entrambe le direzioni. Gli attributi sono single-valued per relazioni binarie nella direzione 1:1 e N:1. Sono collezioni per relazioni 1:N o M:N. Se esistono attributi di relazioni, un costruttore tupla può essere usato per creare una struttura della forma <referenza, attributi di relazione>.

**Passo 3:** Includere metodi appropriati per ogni classe. Questi non sono disponibili dallo schema EER e devono essere aggiunti al progetto riferendosi ai requisiti originari. Il metodo costruttore dovrebbe includere il codice di verifica dei vincoli che devono valere alla creazione un nuovo oggetto. Un metodo distruttore dovrebbe verificare ogni vincolo che può essere violato durante la cancellazione di un oggetto. Altri metodi dovrebbero includere i controlli per ogni ulteriore vincolo rilevante.

**Passo 4:** Una classe ODL che corrisponde a una sottoclasse nello schema EER eredita il tipo e i metodi della sua superclasse nello schema ODL. I suoi attributi specifici e i riferimenti sono specificati come visto nei passi 1 e 2.

**Passo 5:** I tipi di entità deboli possono essere mappati come i tipi di entità regolari. Alternativamente, i tipi deboli che non partecipano in alcuna relazione eccetto che quella identificante, possono essere mappati come se fossero attributi composti multivalued del tipo entità possessore, usando il costruttore set<struct<...>> o list<struct<...>>

**Passo 6:** Le categorie (o tipi unione) creano dei problemi nel mapping in ODL. Si può definire una classe che rappresenti la categoria, ed una relazione 1:1 tra la categoria ed ogni sua superclasse. Un'alternativa è utilizzare il tipo unione, se è disponibile.

**Passo 7:** Relazioni n-arie ( $n > 2$ ) possono essere mappate in un tipo oggetto separato con referenze appropriate a ciascuna classe partecipante. Queste referenze sono basate su un mapping di una relazione 1:N da ciascun tipo di entità partecipante alla relazione n-aria. Le relazioni M:N possono anch'esse usare questa opzione, se richiesto.